ARP: An Efficient Atomic Broadcast Protocol for Cloud Computing Services

Po-Jen Chuang and Wei-Ming Hsu

Department of Electrical Engineering Tamkang University
Tamsui, New Taipei City, Taiwan 25137, R. O. C.

pjchuang@ee.tku.edu.tw

Abstract

For internet servers, how to handle colossal data volumes and maintain data consistency is a major challenge. To ensure each database maintains the latest information and messages do not conflict with one another, we have a new type of database NoSQL (Not Only SQL). A number of atomic broadcast protocols, including libPaxos, Mencius and RingPaxos, have been established to help maintain transmission consistency for cloud computing services. This paper introduces a new atomic broadcast protocol to improve the performance of existing protocols. Our new protocol divides the network topology into various areas to process messages in a distributed way and uses a rotating mechanism to effectively balance the server loads. Experimental evaluation has been conducted to compare the performance of different protocols. The results show that our new protocol yields higher throughput and lower latency under different client loads in contrast to other protocols.

Keywords: *Cloud computing, NoSQL, atomic broadcast protocols, transmission consistency, experimental performance evaluation*

1. Introduction

In today's internet communications, each network service needs to deal with massive traffic loads. Users not only read the information but also become information contributors, producing a large number of writing. To handle the situation and ensure network service quality, internet service providers usually involve multi-layered caching backup mechanisms. How to facilitate data access has now become a critical issue in upgrading system scalability. When facing large-scale, high-concurrency community network services, some choose to manage by powerless approaches, which yet produce other problems, such as requiring higher performance, storage, scalability or availability, to be discussed below.

(1) *high performance*: Massive concurrent reading and writing will heavily increase database loading, like web2.0 applications. How to accomplish high performance to contain practical needs is a major challenge for internet service providers.

(2) *huge storage*: More and more service providers, including community network service websites Facebook and Twitter which dynamically produce a large number of daily users, face the challenge of enormous data storage and access.

(3) *high scalability & availability*: High scalability and availability require horizontal database expansions, which is very difficult to achieve. When users of an application system grow, it is hard for application services to extend performance and load capacity by adding

hardware or service nodes only. Upgrading or expanding is a tough task for sites which need to provide services any time – because they need frequent maintenance and data migration break, and may lose interested users due to lower service quality.

To help internet servers handle the huge amounts of messages, a new type of database NoSQL (Not Only SQL) has been established [1]. NoSQL features *high scalability* and *flexible storage formats*, as opposed to traditional relational databases. When a server corrupts, NoSQL considers how to handle the colossal data to ensure service availability, *i.e.*, to ensure each database maintains the latest information and messages do not conflict with one another. To give an example, Google maintains server information using Chubby [2] under NoSQL (where Chubby adopts the consistency algorithm to attain data consistency between each server).

Atomic broadcast protocols can help maintain transmission consistency for cloud computing services. **libPaxos** is such a protocol based on the message-passing model of consistency [3]. This paper conducts a thorough survey on **libPaxos** and two of its modified forms, **Mencius** [4] and **RingPaxos** [5], to check their advantages and problems. **libPaxos** uses a two-phase commit protocol (2PC) to ensure consistency, **Mencius** improves **libPaxos** by distributing every server's loading with a rotating mechanism, while **RingPaxos** uses the ring topology to advance the tree topology in **libPaxos**. To advance the performance of libPaxos furthermore, we introduce a new protocol in this paper. Our new protocol is called the Area-based RingPaxos (**ARP**) protocol as it divides the topology into various areas to process messages in a distributed way and uses an effective rotating mechanism to balance the loads of servers. Simulation results exhibit that when compared with other atomic broadcast protocols, our new protocol yields constantly higher throughput and lower latency under different client loads.

2. Background Study

Google uses GFS [6] and Bigtable [7] to facilitate internet application services, and employs Chubby [2] to conduct server communications and maintain information consistency for GFS and Bigtable. Chubby enables messages between different servers to retain consistent information by lock service. To achieve such data consistency, Yahoo adopts a similar concept, ZooKeeper [8], which uses atomic broadcast. Atomic broadcast has one problem: how to modify or update a value in the cloud system so that each server can receive a consistent message? A typical scenario is to perform the same operation sequence from the same initial state so that each server can eventually reach a consistent state. To ensure each server performs the same sequence of commands, a consistency algorithm must be executed at each instruction. That is, having a general consistency algorithm which can be widely applied is crucial in cloud computing [9-11].

Mimicking the story about ancient Greek legislators, the Paxos algorithm [12, 13] has its fame in distributed systems due to simplicity and weak, realistic assumptions. It has become a basic element in many fault-tolerant systems or commercial products, such as the Chubby Distributed Lock Service of Google, Analytics and Earth. As the Paxos algorithm can solve the *consensus* problem efficiently, we are now able to build an atomic broadcast protocol to make sure messages are received reliably – in the exact same order by all participants. (No Byzantine problems [14, 15] are assumed.)

The **libPaxos** protocol [3] is mainly about implementing the Paxos algorithm so that people can get familiar with the details. The approach of **libPaxos** is implemented as *a black box* to execute atomic broadcast in distributed applications. The role of a client is to receive and send messages. Other roles are defined, including the proposer, acceptors and learners.

The proposer is responsible for receiving the client message and sending it to an acceptor. It also sends *prepare* and *accept* messages to the acceptor. An acceptor maintains the *client* message in the database, handles the received *prepare* and accept messages, and returns *promise* and *learner* messages. The main job of a learner is to receive the acceptor's final resolution and sort the message order by the round number to check if any message is missing. If a round number is missing in the sorted messages, the learner will ask the proposer to re-send the lost message. Despite being able to solve the data consistency problem and ensure the integrity of each processed message (by using the tree topology to "run the meeting and vote"), **libPaxos** may degrade performance because handling messages which over-concentrate in the proposer requires excessive bandwidth consumption and network loads.

The **Mencius** protocol [4] adopts Paxos and multi-proposers [16] to distribute the proposer loads. It has each server act as a proposer to share the load and, by reducing the excessive load on a single proposer, successfully enhances the overall performance. **Mencius** performs well in distributing the overload on one single proposer, reducing the needed bandwidth consumption, and alleviating the processing bottleneck of proposers (by proposer rotation). It nevertheless inflicts a problem: because the message load and bandwidth are constant for proposers, performance improvement may shrink when proposer rotation is used to avoid the excessive concentration of proposer loads. The **RingPaxos** protocol [5] involves a similar approach as **libPaxos**, except that it uses the ring topology to solve the proposer load concentration problem and as a result increases system performance. **RingPaxos** can effectively distribute the proposer load and contain bandwidth consumption, but when facing an acceptor failure, it has to wait until the acceptor gets repaired in the round, resulting in degraded overall transmission efficiency.

3. The Proposed Protocol

As mentioned, **libPaxos** can maintain desirable data consistency and performance in distributed architectures but needs to send frequent messages from proposers to acceptors and clients. When data volumes build up, it tends to make messages over-concentrate on the proposer, diminishing the effectiveness of the overall mechanism. **Mencius** then brings in the concept of multi-proposers to fix the problem. Besides the multi-proposer concept, our protocol gives a new solution to the problem (of extreme load concentration on a single server): it uses *the concept of partitioning* to handle each message and as a result reduces the affected area to the minimum during server crash – to retain the effectiveness of the overall system.

Our atomic broadcast protocol is an **Area-based RingPaxos** (**ARP**) protocol because it partitions the ring topology into multiple areas. It first adopts **Mencius**'s multi-proposers to reduce the probability of message congestion (on a single server) and then uses *consistent hashing* to implement *the concept of partitioning*. In its operation, **ARP** lets *three* proposers perform the Paxos algorithm in *three* areas, with one acting as the leader to collect the message statistics. To balance the load capacity, the three proposers will take turns acting as the leader proposer. Such a partitioned processing design can evenly distribute the transmission loads and practically enhance the overall effectiveness and scalability of our **ARP**.

Consistent hashing [17, 18], proposed first to solve the hot spot issues in the Internet, is similar to the Cache Array Routing Protocol [19]. The basic principle of *consistent hashing* is that server nodes and keys (based on servers' IPs) are mapped to a $0 \sim 2^{32}-1$ position of the ring structure according to the same hash algorithm. When a write request comes in, calculate the

key IP's corresponding hash value: If the value corresponds exactly to a server's hash value, write directly to the server; if there is no corresponding server, go clockwise to find the next server to write. If finding no corresponding server after hash value 2^{32}-1, re-start from 0. When a found server crashes, continue clockwise to find the nearest server. *Consistent hashing* can effectively reduce the server load and efficiently add or remove any servers, to balance the server load.

As mentioned, **ARP** uses three proposers to execute the Paxos algorithm in three areas. A server is placed under each area to act as an acceptor, responsible for handling a client's request and transmitting it to the proposer in charge (of the area). An acceptor will transmit the *prepare* and *deliver* messages (messages to maintain data consistency); a proposer will handle the *client* messages in its area (received from the acceptor) and send it to the leader proposer. That is, a proposer mainly transmits the *promise* and *result* messages to the acceptor or the *discuss* messages to the leader proposer. The role of a leader proposer will be handling the *discuss* messages received from other proposers and sending the agree messages back to them to ensure data consistency in different areas.

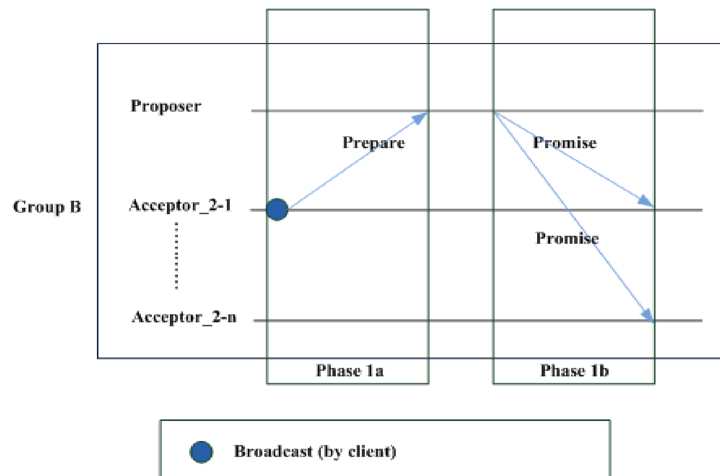

Figure 1. The Process of ARP's Phase 1

ARP divides message processing into Phase 1 and Phase 2. In Phase 1a, as Figure 1 shows, when an acceptor receives a client's request, it sends a *prepare* message to the proposer in charge to check and confirm the round and ballot numbers. This step is taken to make sure the two numbers are not repeated or overwritten, to avoid information inconsistency. After receiving the two numbers, the proposer will confirm if they are consistent with the current running round: If yes, return the latest *promise* message to the acceptor and end Phase 1; otherwise, ask the acceptor to re-send the *prepare* message.

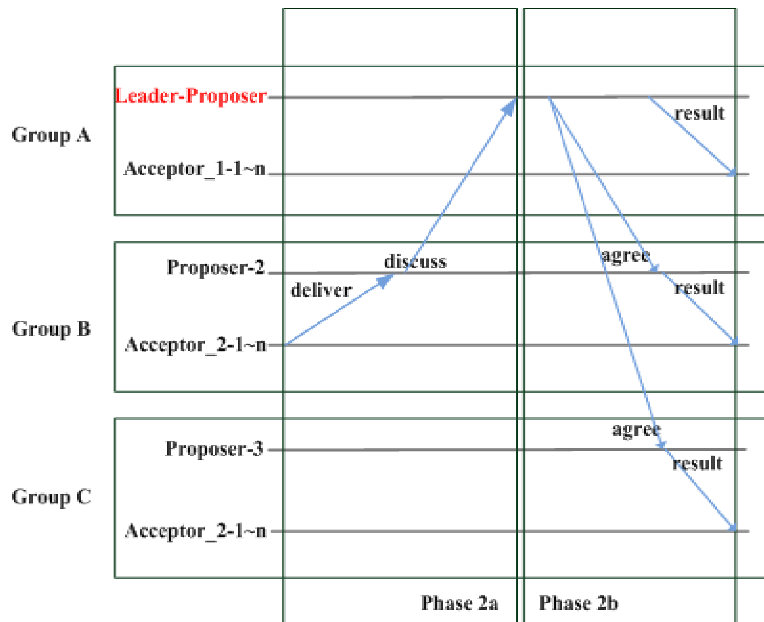

Figure 2. The Process of ARP's Phase 2

In Phase 2a of Figure 2, the acceptor checks the *promise* message (received from the proposer) and sends a *deliver* message including the round and ballot numbers (promised by the proposer) along with the client's request, to the proposer. After receiving the *deliver* message, the proposer sends a *discuss* message to the leader proposer which then compares the *discuss* message with messages from other proposers. If any message conflict happens, the leader proposer will check to see if the received numbers are consistent with the latest values; otherwise, it will proceed to the next step. When the leader proposer sends an *agree* message to the other proposers in Phase 2b, data consistency among different areas is achieved. Receiving the *agree* message from the leader proposer, each proposer will send a *result* message to the acceptor (in its area) which then sends the final resolution to the client.

Table I. Simulation Parameters

	Experiment Setting
Number of values	30 (concurrently)
Value sizes	300、1000、2000、4000 (bytes)
Number of nodes	4
Number of simulation	10
OS	Fedora 8

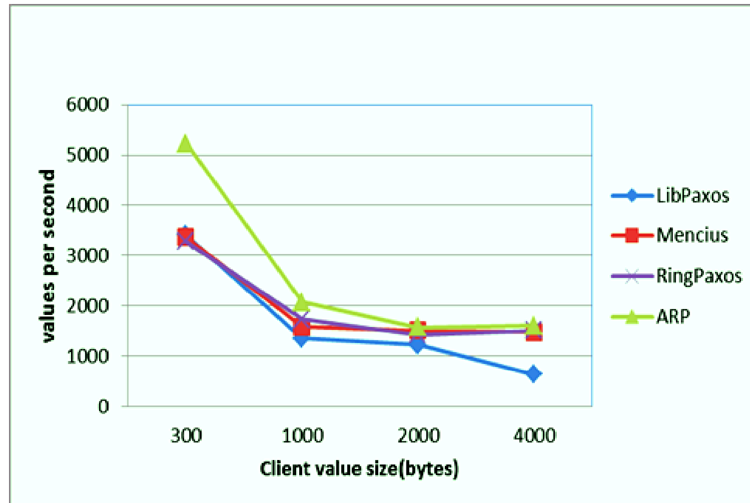

Figure 3. Values per Second vs. Client Value Sizes

Any data consistency algorithms must be able to tolerate the situation of server crash (including the proposer crash and the acceptor crash). In our protocol, when a proposer crashes, the acceptor will wait for the proposer's message until the time limit is reached and then substitute the crashed proposer by a neighboring acceptor. When a leader proposer crashes, the other proposers will wait until it recovers. An acceptor crash will not affect the operation of our protocol because we do not let a client send a request to a crashed acceptor; instead, we will lead the client's request to another acceptor selected by *consistent hashing*.

4. Experimental Performance Evaluation

Experimental evaluation using the DETER Testbed [20] has been carried out to compare the performance of our **ARP** protocol and existing protocols, including **LibPaxos**, **Mencius** and **RingPaxos**. Table I lists the involved simulation parameters. We use *values per second* (*vps*) and *kilobytes per second* (*kbps*) to check the handled data sizes, i.e., the performance, of these protocols. The results are obtained over a large number of values, specifically when clients send concurrently 30 values per 0.1 second. Note that we can measure the data size (*vps* and *kbps*) handled by each protocol from the client side because the client will eventually receive all final resolutions, and the maximal data size a protocol handles can thus be taken as a proper performance indicator.

Figure 3 gives the results of *values per second* (*vps*) under various client value sizes. It shows quite similar results for **Mencius** and **libPaxos** at smaller value sizes when the required processing data and bandwidth usage are containable. When the value sizes grow, we detect degraded processing efficiency (smaller *vps* for bigger value sizes in the figure). This is because the server needs to do extra save when messages increase. When value sizes grow bigger, we see that **Mencius** yields higher *vps* than **libPaxos** thanks to its leader rotation mechanism which keeps messages from concentrating on a single leader. The ring structure in **RingPaxos** helps reduce the load buildup on the proposer and as a result enables the protocol to handle more client messages. At higher value sizes, our **ARP** yields close performance to that of **Mencius** and **RingPaxos** (a result of their leader rotation mechanism or ring structure) but has the best overall performance among all (because we partition the ring topology into various areas and need to handle fewer acceptors in each area).

Figure 4 shows the results of *kilobytes per second* (*kbps*) vs. client value sizes, to examine the message processing ability of each protocol. As we can see, when value sizes increase, **Mencius** and **RingPaxos** work out higher *kbps* than **libPaxos** because they advance **libPaxos** in distributing the proposer processing loads. Our **ARP** outperforms the other protocols. It performs better than **libPaxos** due to the leader-proposer rotation and also than **Mencius** and **RingPaxos** as the result of dividing the ring structure into areas to reduce data transmission among servers.

Figure 5 depicts *latency* (in milliseconds) vs. client value sizes for the four protocols. We define latency as the time duration for a client to send a request and receive the final resolution. Each result here is an average result from 500 samples, *i.e.*, 500 client requests. The results show a reasonable performance trend: When the value sizes grow, latency increases for all protocols. **libPaxos**, as expected, produces remarkably higher latency than others because it has only one proposer to deal with client requests. **Mencius** gives much shorter latency than **libPaxos** because its multi proposers share the message load and speed up message processing. When client value sizes exceed 2000 bytes, **RingPaxos** shows lower latency than **Mencius** – the result of employing the ring structure to carry out message transmission. Among all protocols, our **ARP** yields the least latency. Recall that **ARP** uses acceptors to receive client messages in order to share and ease the burdens of proposers. It can satisfy the needs of clients by using the various roles in the system to equally share the loads of proposers and as a result speed up data processing. As our topology for message transmission basically resembles the ring structure in **RingPaxos**, both protocols generate very close latency when client value sizes > 2000 bytes. But the overall advantage goes to **ARP** – thanks mainly to its special topology partitioning mechanism.

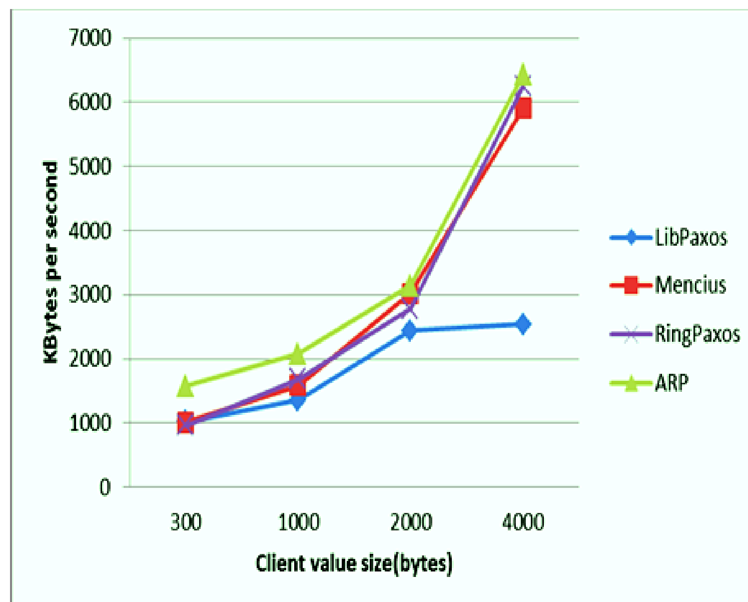

Figure 4. Kbytes per Second vs. Client Value Sizes

Figure 5. Latency vs. Client Value Sizes

Figure 6 gives values per second (vps) under different numbers of acceptors (with the value size = 1000 bytes). As we can see, when the number of acceptors increases, the proposer needs to send messages to more acceptors for resolution, thus degrading the performance. Such a situation will test the scalability of each protocol, i.e., will challenge their ability to maintain the reliability of the entire system. As Figure 6 shows, when the number of acceptors increases, **libPaxos** generates the least vps due to largely increased processing. **Mencius** performs better than **libPaxos** because of the proposer rotation mechanism. With growing acceptor numbers, our **ARP** outperforms the other protocols in most cases because, under *the concept of partitioning*, it lets each proposer deal with their acceptor resolutions only.

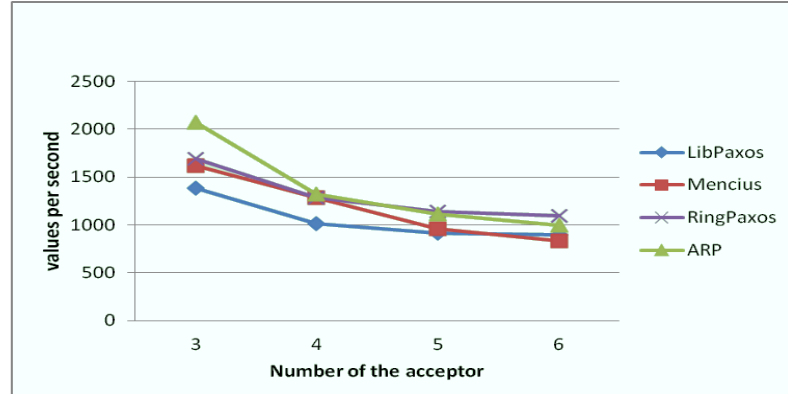

Figure 6. Values per Second vs. the number of Acceptors

5. Conclusions

Atomic broadcast protocols can help maintain transmission consistency for cloud computing services. To ensure transmission consistency for cloud computing services in a more efficient way, this paper introduces a new atomic broadcast protocol, the Area-based RingPaxos (**ARP**) protocol. The distinct features of **ARP** include dividing the ring structure of existing **RingPaxos** into multiple areas to process client messages in a distributed way and using a rotating mechanism to effectively balance the loads of servers. When data volumes build up, messages tend to concentrate on a single server, diminishing the effectiveness of the overall mechanism. To solve the problem, **ARP** first adopts **Mencius**'s *multi-proposer*

concept to avoid message congestion on a single server and then uses *consistent hashing* to implement t*he concept of partitioning*. In its operation, **ARP** lets multiple proposers perform the Paxos algorithm in multiple areas, with one acting as the leader proposer to collect the message statistics. To distribute transmission loads evenly, proposers take turns acting as the leader proposer. This partitioned processing design of our **ARP** can practically balance the transmission loads in the system and enhance the overall effectiveness as well as scalability. As the results of experimental evaluation demonstrate, the above two specific features of **ARP** enable it to outperform existing atomic broadcast protocols in both throughput and latency under different client loads.

References

[1] NoSQL. http://nosql-database.org/.
[2] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems", 7th Symposium on Operating System Design and Implementation, **(2006)**, pp. 335-350.
[3] M. Primi, "Paxos Made Code", Master thesis, Informatics of the University of Lugano, **(2009)**.
[4] Y. Mao, F. Junqueira and K. Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs", 8th USENIX Symposium on Operating Systems Design and Implementation, **(2008)**, pp. 369-384.
[5] P. J. Marandi, M. Primi, N. Schiper and F. Pedone, "Ring Paxos: A High-throughput Atomic Broadcast Protocol", 2010 Dependable Systems and Networks, **(2010)**, pp. 527-536.
[6] S. Ghemawat, H. Gobioff and S. Leung, "The Google File System", 19th ACM SIGOPS symposium on Operating Systems Principles, **(2003)**, pp. 29-43.
[7] F. Chang, Dean, J. Ghemawat, S. W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes and R. Gruber, "Bigtable: A Distributed Storage System for Structured Data", 7th Symposium on Operating System Design and Implementation, **(2006)**, pp. 205-218.
[8] P. Hunt, M. Konar, F. Junqueira and B. Reed, "Zookeeper: Wait-free Coordination for Internetscale Services", 2010 USENIX Annual Technical Conference, **(2010)**, pp. 145-158.
[9] G. Belalem, L. Allal and C. Dad, "Consistency Management of Replicas in Wireless Grid Environment", International Journal of Grid and Distributed Computing, vol. 3, no. 4, **(2010)**, pp. 33-44.
[10] A. Ben Letaifa, A. Haji, M. Jebalia and S. Tabbane, "State of the Art and Research Challenges of new services architecture technologies: Virtualization, SOA and Cloud Computing", International Journal of Grid and Distributed Computing, vol. 3, no. 4, **(2010)**, pp. 69-88.
[11] Y. E. Gelogo and S. Lee, "Database Management System as a Cloud Service", International Journal of Future Generation Communication and Networking, vol. 5, no. 2, **(2012)**, pp. 71-76.
[12] L. Lamport, "The Part-time Parliament", ACM Transactions on Computer Systems, vol. 16, no. 2, **(1998)**, pp. 133-169.
[13] L. Lamport, "Paxos Made Simple", ACM SIGACT News, vol. 32, no. 4, **(2001)**, pp. 18-25.
[14] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, **(1982)**, pp. 382-401.
[15] P. Saini, A. K. Singh, "An Agreement Protocol to exploit and handle Byzantine Faulty Nodes in Authenticated Hierarchical Configuration", International Journal of Grid and Distributed Computing, vol. 4, no. 4, **(2011)**, pp. 11-26.
[16] L. Lamport, A. Hydrie and D. Achlioptas, "Multi-leader Distributed System", U.S. patent 7 260 611 B2, **(2007).**
[17] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins and Y. Yerushalmi, "Web Caching with Consistent Hashing", 8th International Conference on World Wide Web, **(1999)**, pp. 1203-1213.
[18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web", 1997 ACM Symposium on Theory of Computing, **(1997)**, pp. 654-663.
[19] Microsoft ISN: Cache Array Routing Protocol (CARP). http://www.microsoft.com/ISN/whitepapers/cache_array_routing.asp.
[20] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga and S. Schwab, "Design, Deployment and Use of the DETER Testbed", 2007 DETER Community Workshop on Cyber-Security and Test, **(2007)**, pp. 1-1.